PAWAN: A MACH BASED UNIX SYSTEM - (IV)

A Thesis Submitted

in Partial Fulfilment of the Requirements

for the Degree of

Master of Technology

by

Pullela Venkateshwar Rao

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY

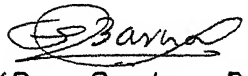
April, 1992

CSE-1992-M
RAO-P

C.S.E-1992-M-RAO-PAW-(IV)

CERTIFICATE

It is certified that the work contained in the thesis titled "PAWAN: A MACH BASED UNIX SYSTEM - (IV)", by *"Pullela Venkateshwar Rao"* has been carried out under my supervison and this work has not been submitted elsewhere for a degree.



(Dr. Gautam Barua)
Assistant Professor
Dept. of Computer Science
and Engg.
I.I.T. Kanpur

April, 1992

Acknowledgements

I am very grateful to Dr.Gautam Barua for his help throughout the course of the thesis.

Just like there can't be a separate unipole, there is no Pull without Mrinal. I can't imagine hawa (PAWAN) without 'Burra' Gopal. Ashi was always ready with a new funda.

I would like to thank all Gulets, Tullas, Ghats, Kushu-shu-bushus and Ahomdas. I thank Impedimenta, Chotu, Desimurgi, Leggie, Ideal, motu, statue, Mscstatshi, Beeta, who kept us (and our tempo) alive at IITK . Thanks to Nayar Hotel at IIT gate for his excellent Dosa and Idli. Special thanks to Govt. (SarkarI) and Deepankar for everything.

Pullela Venkateshwar Rao.

ABSTRACT

In this thesis the design and implementation of PAWAN, a MACH based UNIX system is described. PAWAN consists of a set of user state servers which run on MACH (a multiprocessor based micro-kernel developed at Carnegie Mellon University), and provide a UNIX like user environment. As the first step in PAWAN's development, the source code for the i386 version of MACH3.0 was obtained from CMU and the whole kernel was ported onto the MC68020 based Horizon III (HCL) system in the department. Various distributed services like the file service, naming service, signals, program execution, terminal handling and networking were added to enhance the raw kernel. This thesis contains the details of the file service.

Keeping in mind the large user-group and the vast number of utilities available for UNIX, PAWAN's programming environment was designed to be UNIX source-code compatible with appropriate extensions to UNIX semantics so that they could be used naturally in a distributed multiprocessor system. As of now, PAWAN satisfies all the basic requirements of an operating system. Sophisticated user environments can be provided by porting existing utilities onto PAWAN through minimal effort. Advanced features like network-wide shared memory and load balancing can also be built on the foundation laid by PAWAN. It is expected that PAWAN can fulfill the need for a good developing environment for distributed operating systems at IIT Kanpur.

Table of Contents

List of Figures

CHAPTER 1

INTRODUCTION

The size and complexity of UNIX have increased tremendously over the last few years. With the advent of Distributed Computing and Multiprocessors, the need to incorporate new features in it has constantly been felt. This has reduced the advantages of simplicity and modifiability -- the hallmarks of UNIX during the 1980s. Moreover, there was a clear need to allow the underlying system to be transparently extended to allow user-state processes to provide services which in the past could only be integrated into UNIX by adding code to the OS kernel. These factors led to the development of a new generation of distributed operating systems, for example the Accent, Locus, MACH, Amoeba, V-system and the Sun NFS-based products. MACH (developed at Carnegie Mellon University) is one such operating system which tries to 'kernelize' the functionality of UNIX by providing a small set of primitive functions that allow more complex services to be built as user servers.

This thesis describes the design and implementation of PAWAN, a MACH based UNIX System, which provides an environment for the development of distributed operating systems. PAWAN runs on a network of MC68020-based uniprocessors at IIT Kanpur. It provides traditional UNIX services like file, network, signal and terminal handling through separate user-state servers. This is a significant and novel departure from the single UNIX Server implemented over MACH.

1.1 MACH Operating System

MACH is a multiprocessor operating system implemented at Carnegie Mellon University [Acce86]. The design of MACH was influenced by experience gained with a previous system developed at CMU called Accent. Many MACH features are derived from Accent. The approach in both Accent and MACH has been to design and build a kernel that is suitable for distributed systems and is also able to emulate UNIX. This is quite different from the approach in Locus [Walk83] where UNIX was extended to work in a distributed manner with the resulting system having some of the same limitations as UNIX. MACH is a light weight kernel running in each computer with services such as file system, network service and process management outside the kernel. These services replace the system calls found in conventional operating systems such as UNIX. The model provided by MACH is a service model in which objects are managed by servers, and clients make requests for operations on objects by using remote procedure calls.

MACH has been designed to run in diverse hardware configurations, such as, uniprocessors, tightly-coupled shared memory multiprocessors, loosely-coupled muliprocessor with limited, or differential access to shared memery. It has been implemented for Vax, MicroVax, IBM PC/RT, Perq, Encore, Sequent and Sun machines.

To effectively utilize such machines, MACH provides the following features :

* Separation of typical process abstraction into a task and thread.

* Powerful virtual memory primitives, allowing sharing via inheritance mechanism with copy-on-write implementation.
* A communication mechanism that is transparently extendible over a network.

1.2 Survey of other Distributed Systems

Locus:

Locus is a distributed version of UNIX which provides high degree of transparency of file location and some degree of transparency of location of execution. The features of Locus are system-wide file naming, a file storage system with replication and the ability to run processes remotely. Locus appears to clients like one giant UNIX system, with all of the computers playing both server and client roles and with UNIX file access, process creation and interprocess communication primitives implemented transparently across the network. Relation between the kernel and a user is the same as in UNIX except that the local kernel may route it to another kernel to execute it by using a kernel-to-kernel remote procedure call.

Sun NFS:

Sun's NFS is a distributed file system over which many distributed services like yellow pages are built. NFS provides remote access to conventional UNIX file stores. In NFS, it is possible to mount remote file directories that have been exported by other computers as part of the file name space in local store. Once the appropriate remote file stores have been mounted users and client programs need not be aware of the location of the

files. A user can use the standard UNIX primitives, so programs written to operate on local files can be used with remote files without modification.

The NFS service is implemented in terms of remote procedure calls between kernels. All kernels are both client and server of files and directories. The NFS software consists of a set of extensions to the UNIX kernel and a set of library procedures to enable user-level programs to mount remote files and to export local file systems. The kernel extensions enable a UNIX kernel to act as a client to other kernels when accessing remote files and to act as a server of local files when receiving access requests from other kernels.

Amoeba:

Amoeba is a distributed operating system [Tane90] which runs on a local network of work stations, pool of processors and specialized servers connected to other LANs through a gateway. Amoeba was implemented with many of the components of conventional OS, such as the file service, outside the kernel. The kernel includes facilities for creating processes as clusters of threads, and inter-process communication based on a triple of message passing primitives designed to support RPC messages -

* Request, for use by clients to make remote calls.
* GetRequest and PutReply for servers to receive and respond to server calls.

A user views Amoeba as a collection of processes and information objects maintained by servers. The objects and servers are identified by sparse capabilities. A capability allows a processor to perform certain operations on the object it names.

Several different file systems have been built on Amoeba, including the UNIX file system with UNIX calls, a flat file service and an advanced transaction-based file service called Free University Storage system(FUSS).

V kernel:

The V kernel is a distributed operating system designed for a cluster of computer workstations connected by a high performance network. The design of V kernel relies on the assumption that high performance communication is the most critical factor for distributed systems and protocols [Cher88]. In the V distributed system, separate copies of the kernel run on different machines which cooperate to provide a system abstraction of processes in address space, communicating using a set of communication primitives. The V kernel appears to the applications as a set of procedural interfaces that provide access to the system services.

The V Kernel provides a network-transparent abstraction of address spaces, lightweight processes and fast inter-process communication. Multicasting, a naming protocol and a uniform I/O interface are also provided in the kernel. These facilities together provide a basic framework for implementing variety of services like pipe server, internet server, file server, etc.

1.3 Motivation

At IIT Kanpur, a UNIX compatible OS "IITKIX" was developed over the past few years to experiment with various operating system concepts [Das89]. However, with the advent of distributed computing environments consisting of networks of uniprocessors as well as multiprocessors, we felt the need for a new distributed OS to act as a platform for future research on process migration, distributed shared memory, load balancing and so on. As mentioned earlier, MACH has been designed with the intent to integrate both distributed and multiprocessor functionality. It therefore turned out to be the ideal choice for our purpose.

We obtained the source code of MACH3.0 Micro Kernel for i386 based-systems. Since the hardware description for i386 systems was not available, we decided to port it onto an MC68020-based mini, the HORIZON III. The next step was to build various distributed services in PAWAN as MACH3.0 does not provide a file system, tty i/o, network support or other UNIX features like signals and processes (fork, exec, etc). We decided to build a UNIX-like environment on MACH as the user community is familiar with UNIX, and also the existing UNIX utilities can easily be ported onto PAWAN. We decided for source code compatibility with BSD UNIX with minimal changes as full compatibility would lead to inefficiencies as well as lack of flexibility.

We decided to implement the following servers : Exec Server, File Server, File Pager, Name Server, Network Server, Signal Server and Terminal Server.

This exercise has helped us to enhance our knowledge about UNIX and MACH internals, operating system porting issues and the design and implementation of distributed services.

1.4 Outline of Thesis

Introduction to MACH including its underlying philosophy, abstractions and functions has been dealt with in chapter 2. Chapter 3 describes the overall design of PAWAN, especially its server environment. Issues in porting the MACH3.0 kernel are described in Chapter 4. Chapter 5 describes the file server and Chapter 6 describes the file pager. This thesis concludes with Chapter 7 with the possible future extensions to PAWAN.

PAWAN has been developed in a group at IIT Kanpur. Details of the servers not included in this report can be found in the theses of the other members of this group [Ashi92][Baru92][Gopa92].

CHAPTER 2
MACH OVERVIEW

This chapter presents a brief overview of the MACH kernel. The underlying philosophy, kernel abstractions and features of MACH are examined. More details on MACH can be found in the references [Acce86][Teva87a][Youn87].

2.1 MACH Philosophy

MACH has been designed with a goal of creating integrated computing environments, consisting of networks of uniprocessors and multiprocessors [Teva87b]. The basic functionality of the kernel is designed to support the integrated computing environment of the future.

1) MACH supports diverse architectures (UMA, NUMA, NORMA).

2) It can handle range of communication speeds (LAN, WAN, tightly-coupled multiprocessor).

3) MACH is a new OS organization with

* small number of abstractions.
* kernel/OS server model.
* network transparent and object oriented.
* integrated memory and communication.

The central component of a MACH-based operating system environment is the MACH kernel. Typical operating system services are layered above the MACH kernel as a set of system servers. Since the MACH kernel is a simple base for system servers, it is readily portable and adaptable to the wide range of computing architectures present today and anticipated in the future. Since

servers provide most of the traditional system services, different software environments can be run easily in different hardware environments.

2.2 Basic MACH kernel functionality

MACH can be viewed as being split into two components. The first is the small, extensible system kernel which provides scheduling, virtual memory and interprocess communications, and the second component is the several, possibly parallel, operating system support environments, which provide emulation for established operating system environments such as UNIX.

The MACH kernel supports five basic abstractions: Task, Thread, Port, Message and Memory Object.

- A task is an execution environment and is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as processors, port capabilities and virtual memory).
- A thread is the basic unit of execution. It consists of a processor state necessary for independent execution. A thread executes in the virtual memory and port-rights-context of a single task.
- A port is a simplex communication channel, implemented as a message queue managed and protected by the kernel. A port is also the basic object reference mechanism in MACH. Ports are used to refer to objects; operations on objects are requested by sending messages to the ports which represent them.

- A message is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports.
- A memory object is a secondary storage object that is mapped into a task's virtual memory. Memory objects are commonly files managed by a file pager, but as far as the MACH kernel is concerned, a memory object may be implemented by any object (i.e. port) that can handle requests to read and write data.

Message-passing is the primary means of communication both among tasks, and between tasks and the operating system kernel itself.

2.3 MACH Features

The MACH kernel functions can be divided into the following categories:

- task and thread creation and management facilities,
- virtual memory management functions,
- basic port and message primitives,
- operations on memory objects.

2.3.1 Tasks and threads

MACH divides the typical UNIX process abstraction into two orthogonal abstractions: the task and thread [Teva87c]. MACH allows multiple threads to exist(execute) within a single task. On tightly coupled shared memory multiprocessors, multiple threads within the same task may execute in parallel. The context switching time for threads of the same task is small. Operations

on tasks and threads are invoked by sending a message to the task kernel port and thread kernel port. Threads may be created, destroyed, suspended and resumed. The suspend and resume operations, when applied to a task, affect all threads within that task. In addition, tasks may be created and destroyed. A standard UNIX fork operation takes the form of a task with one thread creating a child task with a single thread of control and all memory shared copy-on-write.

2.3.2 MACH Virtual Memory (VM) Management

MACH has implemented a new, portable memory management system whose main features are:

- Architecture independence: support for a wide range of paged architectures [Teva87a].
- Distributed system and multiprocessor support: features such as shared memory and integrated memory management and message passing [Rash87].
- Advanced functionality: especially, copy-on-write, shared libraries, memory-mapped files and user-implementable memory-objects [Youn87].

MACH Virtual memory interface is divided into several functional groups:

- Address space manipulation including allocation and deallocation of virtual memory of a task at a page level.
- Memory protection allowing flexible use of different memory protection hardware.
- An inheritance mechanism for creation of address spaces in tasks.

- Miscellaneous primitives that formalize access to statistics maintained by the MACH kernel, access other task's virtual memory and describe a task's address space.

The new virtual memory design provides :

- Large, sparse address spaces (needed for large-scale AI application like speech, vision etc)
- Memory mapped files and user-provided storage objects (memory objects) [Teva87d].
- Read/write and copy-on-write sharing (makes possible transparent parallel programming and networking).
- Integrated virtual memory and communication:
    * Large messages sent without physical copies being made (database management, graphics and AI).
    * Memory object capabilities can be passed in messages.
    * Network shared memory with variable consistency constraints.

An important feature of MACH's VM is the ability to handle page faults and pageout data requests outside the kernel [Youn87]. When VM is created, special paging tasks may be specified to handle paging requests. MACH also provides some basic paging services inside the kernel through a default pager task. Memory allocated with no pager specified is automatically zero-filled and its pageout/pagein is handled by the default pager.

2.3.3 Interprocess Communication

The MACH interprocess-communication facility is defined in terms of ports and messages and provides both location

independence, security and data type tagging. Ports are used by tasks to represent services or data structures. Access to a port is granted by receiving a message containing a port capability. Port capabilities include:

- send rights, which correspond to the capability to send a message to a port. Send rights may be held by any number of tasks.
- receive rights, which correspond to the capability to receive a message on a port. receive rights may be held by only one task.
- backup rights, which correspond to the capability to gain receive rights on a port when the task having the current receive rights is terminated. It may be held by only one task.

The MACH kernel automatically queues messages for tasks executing on its machine. However, transmission of messages between separate MACH kernel hosts should be performed transparently by an intermediate server task, known as Network Message Server.

2.4 The I/O structure in MACH

MACH employs an I/O structure substantially different from that of existing systems. Peripheral devices are accessed through the device server port. Device server is implemented as a kernel object. The interface to the device server is through IPC. This section discusses the device server, its services and special features.

2.4.1 Device Switch

Device switch dev_name_list describes the devices potentially attached to the system. Each entry in this table describes the entry points to the driver. Another table Devs gives the bus specific description of devices. Noticeable differences from UNIX are:

* Integration of character and block devices in the same table.
* Kernel just houses the drivers and does initialization. It does not do any other operations on devices.

2.4.2 Normal Input and Output

Device server allocates a port for a device in response to device_open() request. This is the port which is then used to perform I/O on that device. Such ports can then be handed out to various tasks on the discretion of some trusted agent. It may be noted here that send rights to device server port imply complete control over all the devices, whereas access to the port corresponding to a device imply control over that particular device.

Normal I/O is done using MACH IPC on the device port. The I/O can be inband or out of band as suitable for specific devices, inband message passing being more suitable for small amounts of data. Out of band device I/O benefits from copy-on-write scheme used for message passing. It obviates memory to memory copying of huge amounts of data, thus improving efficiency.

Data is not buffered by device server to avoid hard-wiring the policies within it. The applications which use its services are expected to employ their own buffering schemes.

2.4.3 Asynchronous Input

Asynchronous input is handled in a novel way in MACH through a special entry point in the device switch. This entry point device_set_filter(), associates a filter (a boolean function) with a port. The input from device is then filtered and queued at the specified port if filter output is TRUE. This scheme makes it possible to have user level implementations of services which traditionally resided in kernel e.g. network service.

This interface was used in an earlier design of the network server which had a user level implementation. The TTY server uses this interface for filtering special keyboard characters. For example, break key in cbreak mode is provided to the server in the control stream rather than the data stream -- the filter recognizes it and acts in differently.

2.4.4 Iodone processing

Functions performed at the completion of an I/O request are encapsulated in iodone(). In UNIX they generally consist of waking up the process waiting for it and are short enough to be executed from within the interrupt handler. MACH can not afford performing iodone processing in the interrupt routine since it involves data transfer too. Hence iodone_thread is used to perform these functions.

CHAPTER 3

DESIGN OF PAWAN

This chapter describes the design goals of PAWAN, the user state servers in it, and some general issues related to security, user identification, user interface, concurrency and user-state maintenance in servers. The system initialization procedure is mentioned at the end.

3.1 Design Goals

PAWAN uses several user state servers to provide UNIX services in a completely transparent manner. The motivating factor behind this design was our development environment. In a shared memory multiprocessor machine, a single UNIX server implemented through system call and exception redirection (emulation) might prove to be more efficient ([Acce86], [Teva87d], [Youn87]) than our design. However, in a loosely coupled LAN based environment such as ours, a centralized UNIX server which services every user-request, system call or exception in the network will clearly be extremely inefficient. Also, system call and exception redirection by itself is prohibitively expensive on a LAN due to the communication overheads involved. Moreover, MACH itself was envisaged to reduce the number of system calls and hence the number of fundamental concepts for a user to deal with. The only other alternative without using multiple servers is to implement a single distributed UNIX server consisting of several separate UNIX servers running on different machines and communicating with each

other transparently (e.g. LOCUS, chapter 1). Experiences with LOCUS [Walk83] have demonstrated the sheer magnitude of this task and the extreme difficulty in upgrading a huge monolithic system like UNIX.

Another important goal was to make use of existing UNIX code wherever possible (File and Network servers) so that greater effort could be spent in improving the efficiency and modularity of our system. UNIX source-code compatibility has also been a major design goal in the relevant servers since users are used to the UNIX programming environment and existing UNIX utilities could be run without major modifications. However, binary UNIX compatibility is not the driving factor as that would again imply system call and exception redirection which is unsuitable in a loosely coupled system.

We therefore decided to split UNIX services into several independent parts, each of which could be handled by a separate server with an accompanying library. We came up with six user-state servers, the Environment, Exec, File, Network, Signal and TTY servers which are absolutely essential to PAWAN.

1. The Environment Server:

It handles naming in our distributed system by providing a mechanism for tasks to share named variables (ports, strings and environments), an environment being a set of variables. It also provides public and private ports facility to well known servers and is central to System Initialization. It is an extension of the MACH EnvManager and has no UNIX equivalent.

2. Program Execution:

Execve() is implemented as a library routine which facilitates execution of new programs with the help of a resident piece of code which is inherited in a new task from its parent and runs as a co-routine with the main program. The Exec Server intervenes in the execution of setid programs since authentication update is involved.

3. The File Server:

It is a 4.3BSD compatible MACH based file server. It uses 4.3BSD file system code and provides both UNIX-style file-I/O and a pager based interface with external memory management. It can easily be integrated with the network server to provide transparent network-wide file access and shared memory.

4. The Network Server:

It provides a socket based connection oriented and datagram services using the IP suit of protocols. It provides 4.3 BSD socket interface using TCP code from BSD. It is implemented as a privileged kernel task with user interface through system calls.

5. The Signal Server:

It provides a UNIX-like signaling facility. It acts as a forwarding agent for kill requests, and at the same time, it provides means to control runaway or malicious programs. This can be extremely useful in the implementation of UNIX shells.

6. Terminal Service:

The TTY server regulates access to terminals and can service other user interfaces in general. It also provides UNIX like job control, process groups and special character interpretation through signaling. To do so, the terminal driver was modified to use a special filtering mechanism provided in the MACH Device Server. It is separate from the file server.

Possible extensions to this set (e.g., Authentication Server and a Pipe Server) are mentioned in the Conclusion.

3.2 General Issues

In this subsection, we describe some general design and implementation issues common to all servers in PAWAN.

3.2.1 Interface

The following is the description of the interface to all servers in PAWAN:

1. All servers have a well known (public) port to receive user requests and a secure (private) port to receive privileged requests from other servers. All public ports are registered in the Env Server, the Env server's public port itself being available to all tasks by default. Users first query the Env server and obtain the public port of the server they wish to contact, and then send their request to it (FIGURE 3.1).

2. When a server receives the first request from a user task on its public port, it returns a port to the user task for

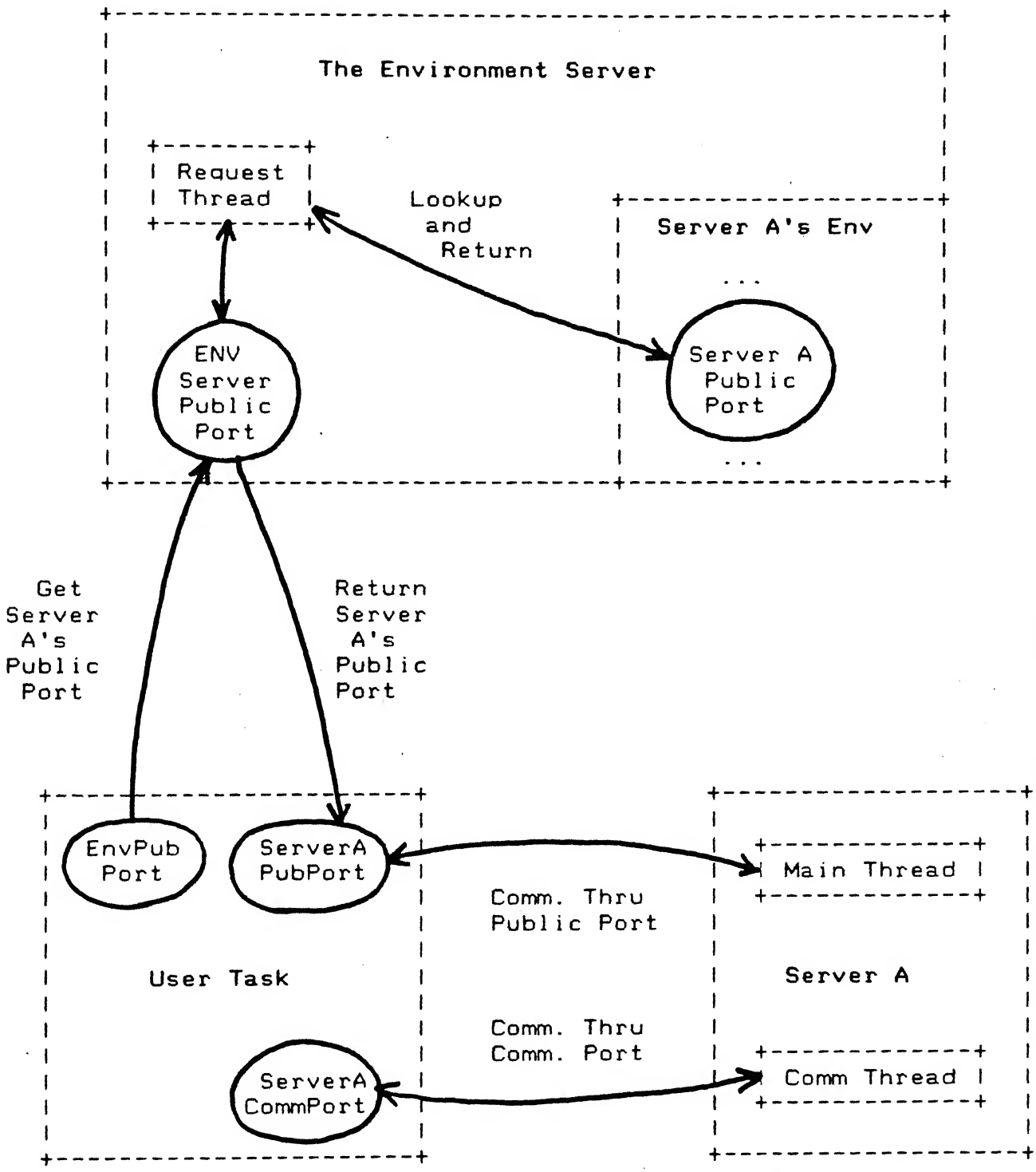


FIGURE 3.1 User Interface to Servers.

communication with it. From then onwards, all future requests to the server can be made on THIS port. Servers have threads waiting on the ports returned above so that multiple requests can be served simultaneously.

3. Servers use their private ports for secure communication among themselves (e.g., changing the privileges of a user task, etc.). These ports are not available to normal users.

4. All requests are handled as Remote Procedure Calls (RPCs) and all servers use the MACH Interface Generator (MIG) [Drav89] in their implementation of the user and server stubs.

3.2.2 Synchronization

In a multi-threaded user-level server environment, synchronization is of utmost importance. In UNIX where all services are implemented through system calls, synchronization could be achieved by simply raising the processor priority level appropriately to block interrupts, since a system call could never be interrupted anyway. At the user level, things are quite different. Modification to data structures shared between the threads of a server have to be done through explicit synchronization through locking variables. This might also involve the use of sleep and wakeup synchronization. The MACH Cthreads Package [Coop88] provides all these primitives at the USER level and has proved very helpful in this context -- most of the servers in PAWAN use it in their implementation.

3.2.3 Unique Identification

UNIX identifies each process by its pid, which is globally unique. In MACH, access control is based on a per-task basis since "the task is the unit of resource allocation". To enforce UNIX-like authentication at any server, we need a per-task-id guaranteed to be unused for a significant period of time after its termination, just as the UNIX pid is. This tid has to be sent in all requests to enable servers to identify and authenticate the user task. The MACH kernel identifies a task by its kernel ports -- but that as a number cannot be used as its tid since its kernel port can be reused within a short period of time after it terminates.

There are two solutions to this problem:

1. Each server uses the send rights of the kernel port of a user task (guaranteed to be unique as long as the server exists) to identify it. The problem with this approach is that each server has a different concept of what the tid of a user-task is (since rights of a port are not unique to all tasks). But as long as rights are transferred between tasks, they get appropriately translated in the new task (as explained in section 5). Some servers (e.g., Signal Server) which provide explicit UNIX services also maintain an integer pid corresponding to every user task since in some cases as the tid is either not enough or unsuitable to identify it.

2. There is a tid-server inside the kernel which automatically generates a globally unique tid for every task

created (just like UNIX). But this unnecessarily increases the complexity of the kernel which is against the basic MACH philosophy -- and should therefore be avoided.

3.2.4 User States in Servers

Servers in PAWAN are stateful, i.e., some state is maintained for every task accessing the server, tasks being identified by their tids or pids (the Env Server uses the env-port to identify an environment since there can be more than one env per user task, and it does not use UNIX style authentication). This state can be maintained in:

1. User space: in which case, the servers have to trust user information and no protection can be guaranteed. Also, every request message has to carry all information about the current user state in it and the corresponding reply has to update it. Though servers can be stateless in this case, the overheads incurred are too high, and this choice was discarded.

2. Kernel space: in which case, either the kernel has to be modified, or the servers have to be inside the kernel -- both alternatives being against our fundamental design goals.

3. Server space: in which case, the user is identified through a (tid, descriptor) tuple and servers become stateful (the descriptor is a communication port or an identifier). This alternative does not compromise security, helps in developing modular user-state servers and is also efficient since it reduces the number of IPC messages. That

This per-task state in each server contains:

1. The authentication information if any (e.g., uids, gids, groups, etc.).

2. The server state corresponding to that task (e.g., ports required for communication with user tasks and some private data-structures opaque to the user).

All servers in PAWAN have been implemented assuming that they know about the death of the user task in some fashion. This information can come from:

1. A notify message from the kernel if the servers request a notification on the death of the user task identified by its tid (= send rights of its kernel port).

2. The exit library routine itself (on a normal termination).

3. The servers can also examine the kernel ports of the user tasks for which states are maintained to determine if the tasks are dead, and if so, do whatever cleanup is required (e.g., deallocate communication ports, free memory, etc.). This can be triggered periodically, or whenever there is a resource shortage.

Other details about state maintenance are described below.

3.2.5 Credentials and Authentication

Every server implementing UNIX-style authentication based on uids and gids needs to have a reliable way to get them for every task in the system (assuming every server has root privileges with uid=gid=0). The credentials of a task can CHANGE

only on an exec (setid) and login. When a task is forked or created through MACH primitives, the credentials only get REPLICATED -- just as the child inherits the uid and gid of the parent after a fork in UNIX. Since user programs are executed either through an exec or a login (through the Exec, TTY and File servers respectively) the relevant servers can initialize the UNIX style authentication information corresponding to a user task so that ALL servers requiring it can access it. This can be done in two ways:

1. That server which first comes to know about the credentials of a task informs all servers using UNIX authentication, of the changes in the credentials of that task (FIGURE 3.2). Each server then updates its information assuming the sender is reliable (since this request is serviced only on its private port). When a parent task does a fork or creates a child task, it is the responsibility of the parent to replicate its state in all stateful servers so that its child can start with the same privileges as itself (UNIX semantics). These replicate requests are not privileged (since the parent can incapacitate only its own child by sending wrong information) and can be serviced for all users on the public ports of the servers.

Hence, in this approach, all stateful servers need to maintain the state of ALL tasks created in the system. Also, if the server has no state corresponding to a tid sent by the user in his request, it is an error on the user's part.

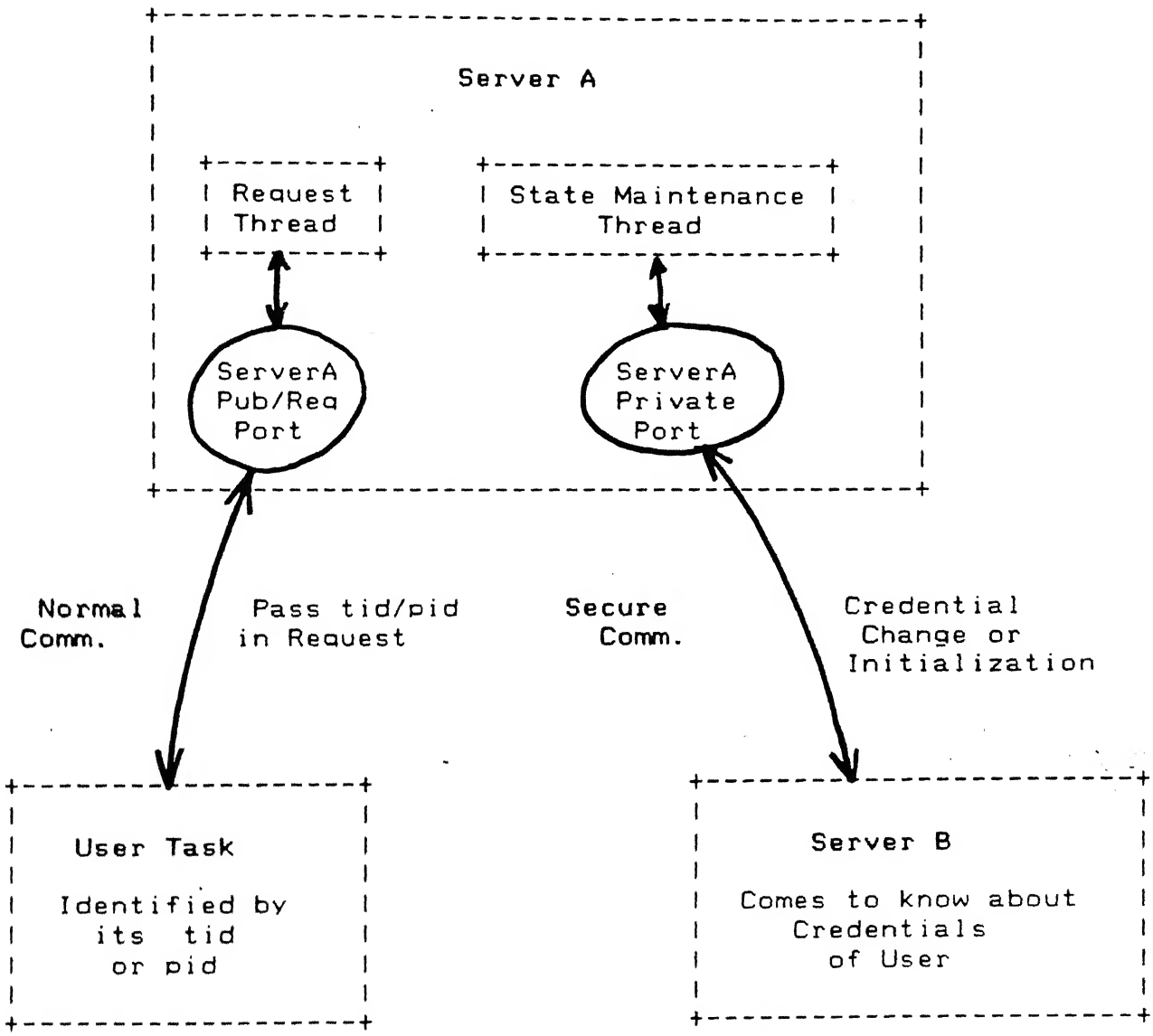


FIGURE 3.2 Authentication of Requests.

The other approach is:

> 2. Have a separate Authentication server which stores a mapping from the tid (or pid) of a task to its UNIX authentication information. Any server which wants to authenticate a request does so by asking the Auth Server. Updates to credential information can be done only by well known servers (request on private port) -- especially the Exec, File and Network Servers. Normal users need to send only their tids (or pids) to servers: servers automatically authenticate them as follows: if a mapping exists in the Auth Server for this tid (or pid), return it, otherwise return an error.

This approach is very elegant and has been used in many distributed systems ([Birr82], [Chas89], [Cher88], [Oppe83], [Tane90]). The advantages are obvious: authentication process is logically separate and hence various forms of protection can be implemented, e.g., Capabilities (login-id and password tuples) or Access Control Lists -- currently not supported in UNIX. Also, servers need to maintain the state of only those tasks which access them, not every task in the system; and states need to be made only when the user task sends the first request to the server. The disadvantage ofcourse is the efficiency of this solution, especially in small scale distributed environments such as ours. In PAWAN, therefore, we have chosen the first approach and have left the Authentication Server as a possible future extension when the size and complexity of the system demand it.

3.2.6 UNIX Compatibility

Though servers in PAWAN adhere to UNIX semantics as far as possible, there are some situations in a multi-threaded environment in which they have to be redefined to extend their functionality and usefulness. Some such situations are illustrated below:

1. Forking a child task: If a task with multiple threads forks a child, should all threads in the task be duplicated or only the thread doing the fork? We should not allow fork in a multi-threaded task due to the obvious semantics and synchronization problems which arise when some threads of the task are communicating among themselves or with threads in OTHER tasks. The same semantics apply to the exec of a file by a multi-threaded task.

2. Fork+Exec of a "setuid/setgid" file: After a fork, the parent has the kernel port of the child. If the child now execs a setuid/setgid file, its file-server permissions change. If the parent now overloads the child task (through a user level exec), the child still has its new privileges but executes a different piece of code -- and if a malicious parent overloads the password program, the consequences can be disastrous. The only solution is: when a user task's permissions are about to change on an exec, the old task is terminated and the exec is done on a newly created task. The parent therefore loses all control of a child which does an exec of a setuid/setgid file.

3. Identification: If user tasks also use send rights of

their kernel ports to identify themselves, then any user task knowing the identity of another user task will automatically have complete control over it and can effectively assume its identity. This scheme is therefore used only by trusted servers. Clearly, the implementation of UNIX-style IPC between unrelated tasks (e.g., signal, kill and wait operations) requires other means of identification (pids) which cannot be used indiscriminately.

There are several other issues relating to UNIX compatibility particular to individual servers. They are discussed in further detail the chapters dealing with their design and implementation.

3.3 System Initialization

The first user program executed by the MACH kernel after kernel-startup is the Environment Server. It creates each server-task, initializes its environment and registers the public and private ports of each server in the environments of ALL servers. Each server-task is then directly executed (without going through the Exec server) by making use of a read only file-system (bootload) implemented inside the MACH kernel. When each server comes up, it initializes the state corresponding to each well known server in the system (identified by the send rights of their private ports). Then, each server waits for user requests on its public port and privileged operations on its private port. Initialization is complete when all servers are ready to receive requests. Users can start interacting with PAWAN when the TTY server gives a prompt and is ready to accept user-input.

The overall view of PAWAN and its server environment, and various important issues in its design and implementation have been discussed in this chapter. This general background is sure to help the reader appreciate the later chapters dealing with individual servers better.

CHAPTER 4
PORTING OF MACH KERNEL

4.1 Introduction

This chapter describes the issues involved in porting MACH3.0 from the original i386 version onto an MC68020-based system, the HORIZON III. This machine was chosen for the following reasons:

1. The hardware description and device characteristics of the i386 machines in our department are not available due to proprietary reasons. Even though CPU-dependent code could run without any modifications as such, it is not possible to run MACH code as a whole without porting the device drivers.
2. However, we had access to three Horizon machines interconnected by the department ethernet. The hardware manuals for the devices connected to it, device driver sources for 4.2 BSD UNIX and the source code for the IITKIX operating system developed for these machines were also available.

Since the Hardware Characteristics of the Source and Target machines are quite different, a lot of time and effort was required to port the MACH micro kernel. In the first step, the machine-dependent portions of the code were identified and their semantics thoroughly understood. They were then rewritten in C and MC68020 assembly language. Most of the time was spent in debugging the new kernel and "fitting in" the changes made in a modular and consistent manner. This task became more difficult

due to the absence of a kernel debugger -- the only debugging-aids available were two routines: siogetchar() and sioputchar() obtained from IITKIX code. These functions access an RS232C serial port of the CPU card connected to the Console and print/wait for a character from it synchronously. Break points had to be inserted manually using these functions and every piece of code had to be traced to ascertain its correctness.

4.2 Target machine description

The HCL Horizon system ([HCLHWM1], [HCLHWM2], [HCLHWM3]) runs BSD4.2 UNIX. It is based on the MC68020 microprocessor with the following peripherals:

1. 16 RS232C serial i/o ports for terminals.
2. 2 100MB disks.
3. 4MB main memory.
4. 1 cartridge drive.
5. 1 parallel printer port supporting Centronics printers.

The MC68020 CPU [MCK20] has eight 32-bit data registers, seven 32-bit address registers, two 32-bit stack pointer registers, a 32-bit program counter (PC), and a 16-bit status register. The status register contains five status flags, three interrupt mask bits, one bit to set either supervisor or user mode, two bits to set trace mode and a bit to indicate that interrupt stack (rather than the supervisor stack) be used for interrupt processing. Memory mapped I/O is used for accessing the peripheral device registers. The CPU can operate in either of two modes -- user mode or supervisor mode (protected or kernel mode).

4.3 Areas of Modification

The following major areas of modification of MACH machine-dependent code were identified:

1. CPU dependent code.

   a) Initialization code specific to the CPU. b) Interrupt and exception handling code. c) Thread and task context switching.

2. Virtual memory management code.

   a) The Pmap module and MMU driver. b) VM fault handling and recovery. c) Page table consistency on context switch. d) Kernel startup and VM initialization.

3. Device Drivers.

   a) Terminal driver. b) Disk driver. c) Network driver.

4.4 Porting Kernel Locore

4.4.1 Exception Handling (Traps and Interrupts)

The MC68020 provides extensive exception processing logic including a very complete set of external interrupts as well as internally initiated exceptions upon detection of various faults, traps, and so on. The internally detected errors are addressing errors, privilege violations, illegal and unimplemented opcodes, instruction traps (trace etc). The externally generated exceptions are bus errors, reset and interrupt request.

Exception Vector Table is central to the MC68020 exception processing sequence. It occupies 1024 bytes of memory.

from physical addresses 0x000000 through 0x0003ff. The table is organized as a 256 four-byte vectors. Each vector is a 32-bit address which will be loaded into the PC as part of the exception processing sequence.

4.4.1.1 Trap Handling

The generic exception handler routine does the following :

1) Save the registers.

2) Determine the interrupt number from the exception frame pushed onto the stack.

3) Call the trap routine to process the individual traps.

4) Restore registers.

5) Return from exception to the original program or abort.

One of the exceptions that needs special servicing is the bus error. A bus error exception occurs when the MMU detects that a successful address translation is not possible. The MACH vm_fault handler routine is called to do the necessary page lookup and update, etc.

4.4.1.2 Interrupt Handling

When a peripheral device requires the services of the CPU or is ready to send information that the processor requires, it may signal the processor to take an interrupt exception. The interrupt execution transfers control to a routine that responds appropriately. The device uses the interrupt priority level signals (IPL0, IPL1, IPL2) to signal an interrupt condition to the processor and to specify the priority of that condition.

The status register of MC68020 contains an interrupt priority mask (bits 10-8). The value in the interrupt mask is the

highest priority level that the processor ignores. When an interrupt request has a priority higher than the value in the mask, the processor services that interrupt. Priority level 7 is the nonmaskable interrupt (NMI), generally assured when power failure occurs.

This interrupt priority level facility (SPL) is also used in the kernel for synchronization and protection of critical sections from interrupts.

4.4.2 Thread Support

A thread consists of a processor state necessary for independent execution. A kernel stack is allocated for each newly created stack. This kernel stack is used for program execution in the kernel mode. In user mode, a separate user stack is provided. When the kernel switches execution among the runnable threads, the kernel stack is also switched and the states of the threads are saved/restored. The processor state of a thread is stored at the bottom of the kernel stack.

The two structures used to describe the various states of a thread are :

struct mc20_kernel_state (thread context) :

This structure (d2-d7, a2-a7, PC, IPL) corresponds to the kernel registers as saved in a thread context switch.

struct mc20_saved_state:

This structure corresponds to the state of user registers as saved upon kernel entry(by traps/interrupts). This is stored in PCB (processor control block) structure in the kernel per-thread structure. It is also pushed onto the stack for exceptions into the kernel.

Save_context() routine saves the thread context and load_context() routine restores the thread context from the base of kernel stack of caller thread. Switch_task_context() routine is used to switch from the currently running thread to a new thread. If both threads belong to the same task, this switching involves only the thread context switching (save/restore the thread state from the kernel stacks). Otherwise, the MMU hardware is updated to refer to the proper page tables (section 4.5).
A new thread is created by a call to thread_create(). The machine-dependent sequence for this new thread bootstrapping can be summarized as :

1) Set up stack & PCB as if the caller had trapped from user space.

2) A dummy frame0 type Exception Frame is created in the stack.

3) return from the kernel as if returning from an exception.

4.5 Porting Virtual Memory

Virtual Memory in MACH has been designed in such a way that its primitives are simple and general enough to be implemented on any paging system. This is best reflected in the fact that all the machine dependent portions are completely separate from the machine independent parts. MACH VM consists of four components: Resident Page Tables to manage physical memory, Address Maps to manage non-overlapping ranges of virtual addresses of tasks, Memory Objects which provide means for external memory management, and Physical Maps ("pmaps") which are

machine dependent software structures corresponding to the address maps. For the Horizon III, these correspond to the hardware page tables. Porting MACH VM therefore primarily involved porting the Pmap Management module which gives a machine independent interface to the rest of the VM code.

Before MACH VM could be integrated with the rest of the kernel, we had to:

1. Develop a specialized MMU driver to facilitate the implementation of the Pmaps.
2. Modify resident page table management routines to take care of paging of page table pages.
3. Implement routines to manage page tables referred to by the MMU during the context switch of threads.
4. Implement the VM initialization code which is executed on kernel startup.
5. Write the VM fault handler executed from the low-level trap module ("locore") to handle validation and protection violations in both kernel and user modes. Special attention was given to recovery from vm-faults in the kernel mode.
6. Write various miscellaneous routines required to integrate VM and IPC (e.g., copyinmsg(), copyoutmsg() among others).

Though the modular implementation of MACH VM helped us a lot, we had to take special care to maintain the hardware consistency of virtual to physical mappings whenever they were updated or deleted, especially during the context switch of threads not belonging to the same task, and during switching of the processor state from the kernel mode to the user mode and vice-versa.

4.6 Kernel Startup

Here we describe how the PAWAN kernel comes up. First, the kernel is bootstrapped enough to run with virtual memory since at the time the PROM loader loads it onto the physical memory, virtual memory is not up and only physical addresses can be accessed. Bootstrapping involves creating a fixed number of kernel page tables at the end of the text and data regions in the physical memory and mapping virtual memory corresponding to the whole of the physical memory until some maximum virtual address. This completes the machine dependent part of VM initialization.

The resident memory module which manages the available physical memory (starting from the end of the kernel page table pages and going upto 4MB) is initialized at this point. Once the address map module and the memory object (pager) module are up, the kernel is ready to manage VM. Interrupt Vectors are then loaded and devices are probed and started up. This is followed by the initialization of task and thread management portions of the kernel after which the timer is started to enable scheduling. The rest of the kernel including IPC and network management is now initialized in a machine independent fashion. Finally, control is transferred to the first user program, "startup".

4.7 Porting Device Drivers

MACH device drivers are similar to their UNIX counterparts. In fact some of the device drivers for PAWAN have been picked up from IITKIX, an experimental thread based

operating system [Das89]. Rest have been coded afresh. Though there are changes in the way information regarding device addresses and Q-bus structures are allocated, the functionality remains the same. Another addition has been that of asynchronous input interface to the network and tty drivers.

4.8 PAWAN Development Environment

The PAWAN development environment consisted of a collection of SUN-3/60 workstations running SUNOS, connected by 10 Megabit ethernet. The target machine was HCL Horizon III which is a MC68020 based minicomputer. The decision to use Sun systems was taken because they had ample disk space, excellent windowing systems (sun windows and x-windows) and more processing power.

The kernel and all other user level programs were compiled on Sun systems and the object code was transferred to the Horizon machine using "rcp" and "ftp" services. Such an arrangement was feasible because the object code produced on Sun machines could be run on Horizon. This scheme speeded up the development process by a significant amount due to the parallelism it provided -- any member of the group could debug on the Horizon (by booting it) without precluding editing or compilation by others.

GNU software was used throughout for language translation and program management as MACH source code required such support. Revision Control System (RCS) available on the Sun was used to manage the huge amount of source code of our system.

CHAPTER 5

FILE SERVER

5.1. Introduction :

This chapter describes the file system support provided in PAWAN. As MACH is a micro kernel and provides no file system support, a 4.3 BSD compatible file system has been designed and implemented as a user state server. MACH kernel has a stand alone loader and a default pager. The loader has a read only file system with no buffering and provides services to kernel tasks only. We need a full fledged file system which can cater to the needs of the kernel as well as user tasks. We wanted it to be compatible with UNIX so that large number of available UNIX-utilities can easily be ported on PAWAN. Our User File Server (UFS) runs along with other servers in the system and uses the kernel device server for device I/O. As the source code for BSD is widely available, we modified BSD file system source code to make it a separate the file server. The Cthreads Package is used to write file server [Coop88].

5.2. Design goals :

Major design goals of our file server are :

- Compatibility with UNIX file system for easy porting of UNIX utilities.
- BSD 4.3 source code to be ported with minimum possible modification.
- Adding paging support should be easy.
- It should exploit MACH features like copy-on-write to

minimize multiple copies of data, should copy VM-maps rather than actual data and use threads so that a task can do multiple file operations simultaneously.

- It should be a user state server without compromising security
- It should retain device support through file system for future use and possible extension to support all special files.

5.3. Design and Implementation :

In BSD, user state is maintained in U area inside the kernel. In PAWAN, UFS maintains the file related state of users. UFS is a user state task with many threads of control viz, master thread, read-write(RW) threads, U area maintenance(UM) thread. Client tasks send file open requests to the master thread. This thread returns a port as a file descriptor after opening the file. Read and write requests for the open file are sent to that descriptor port. A read-write(RW) thread waits on the descriptor port for servicing these client requests. Design of U area in UFS is explained in section 5.3.1. Section 5.3.2 describes the UFS threads. Section 5.3.3 describes the interface and working of UFS threads. Other modifications done to BSD are described in section 5.3.4.

5.3.1 U area :

U area in BSD is defined as : a per process structure maintained in the kernel that contains process state information that is not needed in core when a process is swapped out[Bach86].

It contains arguments to system calls, system call return values, uid and gid of the process, open file descriptor tables, namei cache, pointer to proc table, signal management data, timing and profiling data, quotas etc.

A user task in MACH typically contains multiple threads which can request file system services simultaneously. To service these requests in parallel, U area has to be split into a per request U area(U-request) and a per task U area(U-task). U-request is placed in a per request structure called file server request(fsr), which includes many other parameters like file mapping information,offests etc. U-request contains parameters, return values , error value etc. U-task structure contains the fields common to all threads in a task. For example , UID, GID, current directory etc.

5.3.2 Description of UFS threads

In UFS, master server thread comes up first. This thread does initialization of UFS data stuctures. It opens the root device and gets the super block information to mount the root file system on '/'. It forks the UM thread and then waits for requests from users on UFS's public port.

UM thread in UFS maintains the per task U area (U-task) of all tasks in the system. When a new task comes up in the system, Exec server requests the U area maintenance(UM) thread to create a U-task for the new task (Refer to Fig 5.1). Along with this request, it also provides the kernel port of the new task for identification. UM thread creates a new U-task and associates it with the kernel port of the user task. When the new user

task wants to use the file server for opening a file, it sends a request along with its kernel port to the master server thread. Master server thread will hash the kernel port to get the U-task of the user task. UM thread also takes care of cleaning up of U-task when a task exits in the system.

UFS maintains one read-write thread per open file. They are created by the master thread when a file is opened and they are destroyed when the file is closed or the user task has exited. They service all requests on an open file like read, write, fsync, ftruncate, close etc. Any calls that are general to UFS are serviced by the master thread. For example, calls like sync, chdir, chroot, mount etc.

5.3.3 The protocol :

Interaction between various UFS threads and user threads is shown in Fig 5.1 to Fig 5.6. User threads send requests for opening a file to the public port of UFS. Master thread waits on this port for servicing these requests. Apart from file name and mode, an open request includes the kernel port also. The kernel port is used to identify the user task.

Master thread opens the requested file and obtains a file descriptor. It allocates a per open file request structure(fsr) and stores the descriptor in the new fsr. A new port(descriptor port) is allocated and a RW thread is forked (refer to Fig 5.2) which waits on this port to service all requests on this open file. This port is associated (hashed) with new fsr to identify the fsr corresponding to a descriptor port. This port is returned to the user task as a file descriptor. User thread sends

1) ES informs UFS of a new task

   Ucreate(kernel_port ...)

2) Create U-task for new task

3) open(kernelport,fname,...)

4) hash kernelport to get U-task.

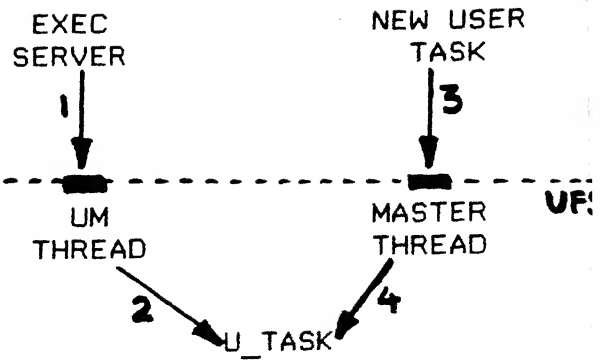


figure 5.1 : Init

5) Create descriptor_port

6) Fork a RW thread to wait

   for requests on it.

7) Create fsr

8) return descriptor_port.

9) read(kernel_port,

   descriptor_port, ...)

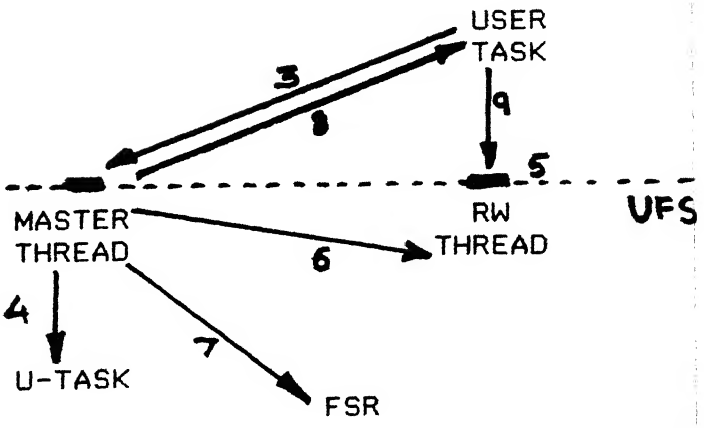


figure 5.2 : Open

10) hash kernel_port

    to get U-task.

11) hash descriptor port

    to get fsr.

12) send data to user.

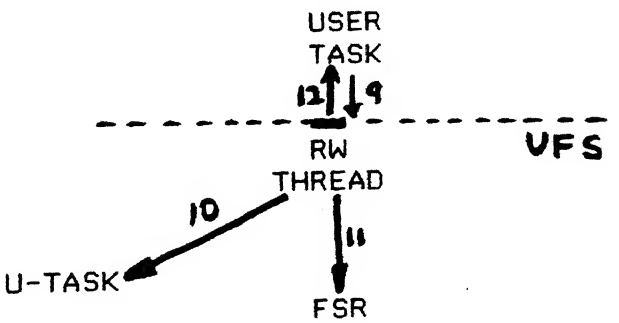


figure 5.3 : READ

13) Ureplicate(child_kernel_port,
parent_kernel_port, ...)

14) Parents U-task replicated
to U-task'.

15) read(child_kernel_port,
descripor_port, ...)

16) Child_kernel_port is hased
to get U-task'.

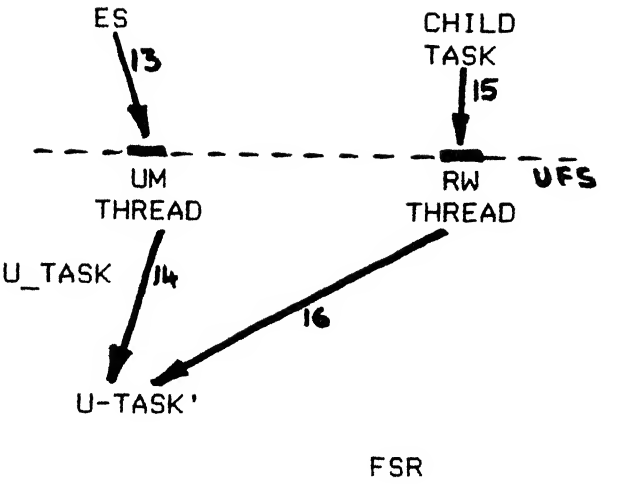


figure 5.4 : Fork

17) Close(descriptor_port, ...)

18) Clean up fsr.

19) RW thread destroys descriptor
port to kill itself.

20) RW sends reply to user before
exiting.

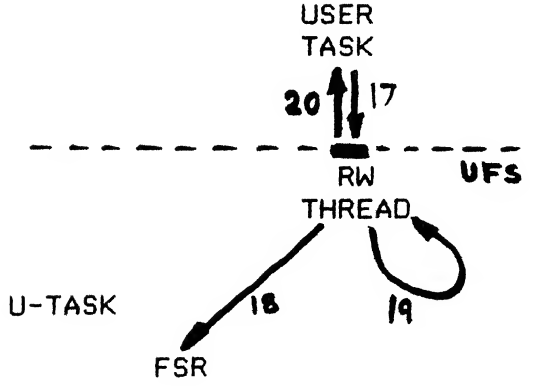


figure 5.5 : Close

21) User task's death is informed.

22) Deallocate U-task.fsr.

23) Kill RW threads.

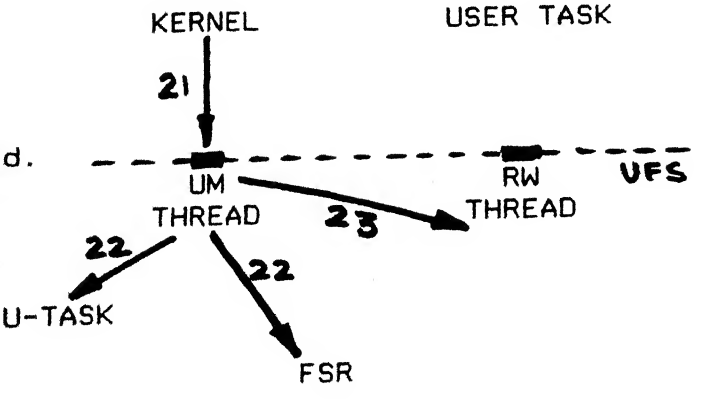


figure 5.6 : Task Death

all requests for operations on an open file to the file descriptor port (refer to Fig 5.3).

Operations that are not done on open files like chdir, sync, cmask etc are serviced by the master server thread. It uses a temporary fsr structure and the U-task of the user task to execute these calls so that any change in the current directory or mask value etc, goes to the user task's U-task and, parameters, return values to a temporary fsr.

Child tasks created through fork inherit open files of its parent task. It should send requests to the same descriptor ports as the parent's (see Fig 5.4). UNIX semantics are maintained in fork as well as in exec. Exec server informs UM thread about the task fork and specifies the kernel port of the child task. UM thread replicates the U area of the parent task and increases the reference counts of all open files. Child tasks kernel port and new U area are associated. Requests on a particular open file come to the same RW thread from both the parent and child tasks. This file is kept open until both child and parent close the file. When a request for operation on a open file comes from a child task, RW thread hashes the the child kernel port to obtain child task's U-task to service the request. This is done transparently as child and parent tasks give their kernel ports which are used to get their corresponding U-tasks and to service the requests in their respective U-tasks.

A file is closed if a user thread specifically requests for it or if the associated task dies without closing the file. If a thread specifically requests for closing a file, the RW

thread hashes the descriptor port to get the fsr structure of the file (illustrated in Fig 5.5). If the reference count in fsr structure is more than one (in case of a task fork after opening the file) then the reference count is decremented by one. If it is equal to one (no other user is using this port for file operations) association between fsr and descriptor port is removed, and descriptor port, and fsr structure are deallocated. The kernel informs UFS about a task's death in the system as it has send rights to the kernel port of all tasks in the system. If a user task dies before closing opened files, they are closed by UM thread. UM thread deallocates the U-task of the dead task and deallocates all the open file descriptor ports of that task by killing all the RW threads waiting on these descriptor ports. All open file descriptor ports are stored in U-task for this cleaning up operation. These are updated when a user thread specifically closes any file.

5.3.4 Miscellaneous Changes to BSD :

5.3.4.1 U area passing :

BSD file system executes in user's context. Any reference to variable 'u' will be referred to the U area of the user [Leff89]. To simulate this, fsr structure is passed to every BSD routine as there are no per thread global variables. As a large number of software developers are familiar with BSD source code, we tried to maintain similarity between BSD code and file server code by leaving the BSD source code unchanged until compilation time, where ever possible. Before compiling the source code it is

passed through a filter which will replace all U area references with fsr structure and corresponding fields.

5.3.4.2 Memory allocation :

Buffers are allocated 8kb of virtual memory and 2kb of physical memory in BSD. The buffers which need more physical memory to service requests larger than 2kb will take physical pages from other free buffers. This is done by manipulating page table entries in BSD [Leff89]. In UFS, all buffers are allocated 8kb of virtual memory. Physical memory is allocated by the kernel as and when the pages are referenced.

5.3.4.3 Device server interface :

Device interaction in BSD goes through bdevsw(block device switch) and cdevsw(character device switch) tables. These tables contain pointers to device driver routines for all devices in the system . In PAWAN, device service is provided by a separate device server task inside the kernel. Stub routines are written to convert device driver requests into remote procedure calls to the device server. UNIX device interface is retained for easy porting of UNIX programs i.e., all special files like /dev/console, /dev/ttyh0 can be serviced through UFS.

5.3.4.4 VM operations and Copy-on-write sharing :

In BSD, memory is allocated by the kernel and a device driver simply copies data to and from this memory. In MACH, the device server allocates memory in client address space for every read request. A client should deallocate this memory when the data is no longer used. Hence the stub routines in UFS that interface

with the device server copy data using vm_copy and then deallocate the data returned by the device server.
In response to a user read request RW thread allocates memory using the vm_allocate system call. Requested data is copied from the buffer cache into this memory and returned to user address space through IPC. Virtual memory operations vm_allocate, vm_copy etc are used to speed up copying. All these virtual memory operations operate on page table entries unlike their BSD counterparts which copy data byte by byte. Any data sent through IPC is shared copy-on-write between sender and receiver i.e., they share the same physical pages as long as they don't write into them.

Wiring of pages i.e., preventing paging of physical pages is allowed to only kernel tasks in MACH. Because UFS is a user state task its memory can be paged out just like any other user task. This slows down UFS slightly if memory is used to its capacity.

5.3.4.5 Synchronization , sleep and wakeup :

BSD does not allow a context switch when a system call is being executed [Bach86]. This is to maintain consistency of the kernel data structure. Since UFS runs as a user task, a RW thread may be preempted and another RW scheduled while the first thread is manipulating the global data structure(critical code). To avoid mangling of global data, threads in UFS cooperate within themselves and maintain the consistency of the global data structures. A mutex(mutual exclusion) variable exec_mutex is used for this purpose. Any thread wanting to execute has to lock this

exec_mutex first. Only one thread can lock this mutex at a time and other threads are blocked trying to lock it. This ensures that only one thread executes the critical code. Locking of this mutex is confined to the critical code only to allow threads to execute non critical code simultaneously.

Condition variables buf_free, ino_free are used for sharing of buffers and inodes in UFS. Buffers are locked by setting a flag in the buffer by the thread that is using them. Any thread trying to use a buffer checks whether it is locked or not. If it is not locked the thread locks it and goes ahead. If it is locked, the thread does a condition_wait on buf_free condition variable. Threads do a condition_signal on buf_free when unlocking a buffer, waking up all the threads waiting for buffers. The code for this looks similar to sleep wakeup code in BSD. Condition variable ino_free is also used similarly.

```
while (bp->b_flags & B_BUSY)
        bp->b_flags |= B_WANTED;
        condition_wait(buf_free, exec_mutex);

. . . . . .

bp->b_flags &= ~B_WANTED;
condition_signal(buf_free);
```

In BSD, interrupt priority level is increased to prevent interrupts when executing critical sections of code. This is because ISR may find the kernel data structures inconsistent as they are being modified. This problem does not arise in UFS.

5.3.4.6 User stub and server stubs

Interface to the file server is written in MIG interface language. This is compiled to get the user and server stubs. Apart from these stubs two more stubs are written to take care of parameters and buffering, size of I/O etc. First one is on server side which receives and passes parameters appropriately to BSD routines. This also checks the return value and sends the appropriate return code to user. This resembles the system call entry and return code in BSD.

Similarly a user stub can be written which does buffering of file I/O. That is any read is done in multiples of page size of the system and the read contents are buffered. Stdio library can be easily made a user stub to give this standard UNIX I/O library.

5.4. UNIX compatibility :

Though we started with the goal of providing a file system mostly compatible with UNIX, we achieved total compatibility. The file server provides all system calls in BSD 4.3 UNIX and stdio library can easily be ported. Device interface through file server is retained for future use. File locking and select are not implemented.

CHAPTER 6
FILE PAGER

6.1. Introduction :

This chapter describes the design and implementation of a file pager which provides mapped files in PAWAN. A mapped file is an address range where a file is mapped, which can be accessed just like any other region of memory [Teva87d]. Mapped file facility allows a user to treat file data as normal memory without regard to buffering or concerns about sequential versus random access. Section 6.2 describes the external memory managers and memory objects. Section 6.3 describes our implementation of FP. Section 6.4 describes the protocol used for external memory management. Section 6.5 explains the uses and the problems with pagers.

6.2. External memory management

6.2.1 Memory managers :

In MACH user state tasks can act as memory managers [Youn87]. Memory manager is a server that provides memory management services like paging. MACH kernel has a default memory manager which does paging for kernel provided memory objects. User tasks can act as memory managers to special objects like fault-tolerant objects, objects whose backing store is across networks or objects whose backing store is on devices for which the kernel does not provide drivers. The protocol used for external pagers is shown in Fig 6.1 to 6.3 . Memory manager tasks have more

responsibility than client tasks and should respond to all the kernel requests, otherwise threads in kernel and client task that are attempting to reference the memory object will be left hanging. It is also possible for a privileged user to replace the default memory manager. This involves increased reliability and responsibility as now all the users of the system will be dependent on the new server.

6.2.2 Memory objects :

In MACH physical memory is used as a cache of the contents of secondary storage objects called memory objects. Virtual memory of a task is a series of mappings between address ranges to such memory objects. For each memory object kernel keeps track of those pages that are in cache and it allows the tasks mapped to that memory to use those physical pages. When a virtual memory location is referenced whose data is not in cache, kernel should make a request to the memory object for the required data. Kernel should also write back any changes done to the data to corresponding memory objects. A task may map a specific memory object into its address space by issuing a vm_map call. This request includes a memory object port which will take responsibility of servicing kernel requests for read and write to the object. A given memory object may be mapped into many tasks. Kernel recognizes the memory object if it is mapped before. Physical pages for this memory object can be shared by all the mappings. The MACH kernel keeps physical memory cache consistent for all uses of a memory object. A single memory object may be mapped into tasks on two different hosts. In this case memory

manager has to maintain consistency among various hosts on which data resides.

6.3 File Pager (FP) :

Our File Pager is a memory manager that does paging for files. This services the kernel requests for file data in a 4.3 BSD file system. Pager interface is provided by separate pager threads which run along with other threads in the same UFS task (described in chapter 5). Pager and file server are kept in the same task to minimize message transfers.

6.4 The protocol :

The protocol used for mapping a file consists of three parts. First are the calls made by the user on file pager(FP) and kernel to map a file into its address space. Second are the calls made by the kernel to the file pager. Third are the calls made by the pager to the kernel.

6.4.1 User calls :

User sends a file map request to FP requesting it to be the pager for a file (refer to Fig 6.1). The parameters included in this message are file name, mode of mapping, offset in the file where the mapping should begin, size of the mapping. FP returns a memory object port if user has proper access permissions for the file. User does a vm_map system call requesting the kernel to map the memory object into its address space. The port returned by FP is sent in this call for kernel to refer to that memory object.

1) file_map(filename,...)

2) returns memory_object(a port)

3) vm_map(memory_object,address,
offset,size, ...)

4) memory_object_init(memory_object,
page_size, ...)

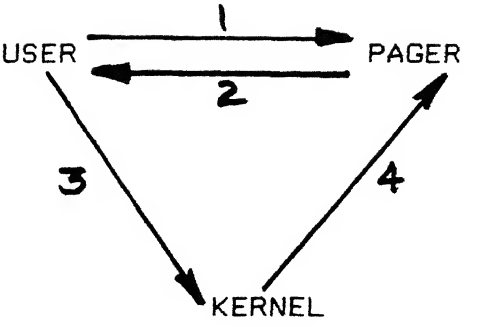


figure 6.1 File map

5) Page fault on accessing address

6) memory_object_data_request(
memory_object, offset, ...)

7) memory_object_data_provided(...)

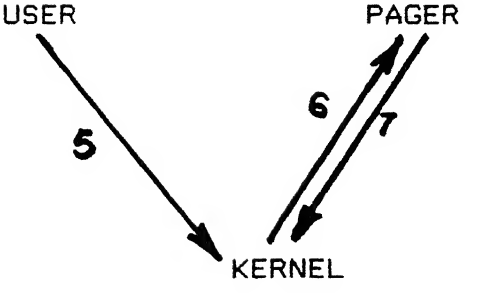


figure 6.2 Page fault

8) user exits or
deallocates address.

9) memory_object_terminate(
memory_object)

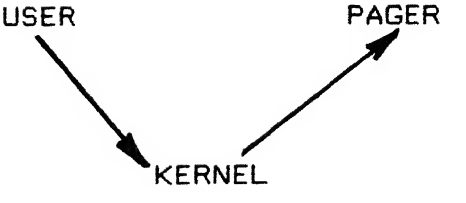


figure 6.3 Cleanup

6.4.2 Kernel and file pager calls :

This protocol is defined by MACH for interaction with all external pagers [Youn87]. All remote procedure calls made by the kernel and pager in this interface are asynchronous

When asked to map a memory object the kernel responds by making a memory object init call on a memory object port. This message includes a memory object request port which the pager may use to make cache management requests to the MACH kernel and a pager name port which the kernel will use to identify this memory object to other tasks. In order to process a cache miss(i.e., page fault), the kernel issues a memory_object_data_request call specifying the range(usually a page) and the pager request port to which the data should be returned (refer to Fig 6.2). To write back dirty pages the kernel performs a memory object data write call specifying the location in the memory object and the data to be written. The pager passes data for the memory object to the kernel by using the memory_object_data_provided call including location of the data in the object.

A typical pager provides data only on demand (in response to memory object data request). However a pager may provide more data than requested. Since a pager may have external constraints on the consistency of memory objects, MACH kernel provides calls to control caching. The cache consistency calls are not used by FP as it does not provide sharing of mapped files by two user tasks.

A pager may restrict the use of cached data by issuing a memory_object_lock request, specifying the types of access(read,

write, execute) that must be prevented. If greater access is required to cached data the kernel issues a memory object unlock call . The pager is expected to respond by changing the locking on that data when it is able to do so. When no references to a memory object remain and all modifications have been written back to the pager, the kernel deallocates the memory object request port. Pager receives a notification of port destruction, at which time it can perform appropriate shutdown (refer to Fig 6.3).

6.5 Kernel created memory objects :

When a task does vm_allocate call the kernel allocates a memory object that provides zero filled memory on reference. This object is managed by a default pager.

CHAPTER 7

CONCLUSION AND EXTENSIONS

7.1 Conclusion

The work described in this, and companion theses, lays out a platform for future work, analogous to a hardware backplane on which different cards can be plugged in. Our basic aim was to experiment with a system which could survive in a rapidly changing world. As research has brought out, an extensible system with user state servers is the solution. This thesis has discussed design and implementation of File server and pager on PAWAN.

We strived to investigate and pin down the specific issues and trade-offs which govern the characteristics of an operating system in a scalable, multiprocessor and distributed environment. Efficiency was not our immediate concern (none of the servers use elaborate searching schemes). We believe that with a proper design, efficiency issues can be tuned up in later stages of development. To this end, we have been able to meet our goals, which have been listed below:

* Isolating hardware dependencies in porting and mapping hardware characteristics of various machines to enable quick porting.
* Providing a workbench for experimental distributed system research.
* Exploiting machine and operating system features to improve flexibility and to encourage new programming

methodologies (e.g. having multi-threaded servers).

* Designing a communication oriented system to exploit parallelism.

* Designing a unix-like system which can be transparently extended using outside kernel solutions, yet maintaining UNIX source code compatibility with minimal modifications.

UNIX was designed for an isolated, uniprocessor system and it flagrantly falls short of expectations in different kinds of system architectures which are prevalent today. One facet of our work was to isolate instances where UNIX methodology is not the best one to pursue. Appropriate solutions, and more importantly principles involved, have been worked out to tackle such intricate issues.

Since users are more used to UNIX and there is a vast amount of source code existing for the same (not to mention the unix associations), providing a unix-like system was one of our goals. As pointed out at various occasions earlier, UNIX semantics are difficult to achieve in newer systems. In fact it is even inappropriate or undefined in some cases (e.g. forking a multi-threaded program). Redefining UNIX semantics to suit PAWAN design and simulatneously being benign to the old programs and new users has been a major achievement on our part. A pertinent example is the subtly changed behaviour of signals.

PAWAN, at the time of writing of this thesis, is not a complete system in itself. It provides most of the OS services, namely a file system, program execution, global name space in the form of environment manager, a usable interface to the network

and UNIX style signals.

A sophisticated user environment does not exist -- programs have to be edited and compiled either on other systems or in UNIX environment of the same machine. Also, there is no shell.

We have not been able to gather performance statistics due to lack of time. It is our conviction that efficient design and implementation together with accurate OS primitives will dominate the context switch overhead, thus gaining in performance over BSD4.3. Future theses on PAWAN will have more to say about it.

7.2 Suggested Extensions

The two aspects of the extension needed are -- to provide a user environment and to add tools enabling extensive distributed and multiprocessor system research.

To provide a program development environment, shell, compiler and editor must be ported. This should be easy as fundamental libraries involving file system, signals and network have been provided. Other utilities can also be ported in an incremental manner. A more comprehensive environment will require more servers e.g. pipe server and authentication server.

The other aspect viz. tools for distributed and multiprocessor system research, is more important. There is ample scope for innovation here as these are not going to hinge upon UNIX compatibility.

Following extensions can be immediately sought:

* Providing full range of network services including network monitoring.
* Implementing higher primitives -- distributed shared memory and net message server.
* Implementing a processor server to experiment with multiprocessor scheduling.
* Identifying issues in process migration and load balancing.
* Incorporating various I/O scheduling strategies in device server.

An entirely different research direction could be that relating to fault tolerance in distributed systems. Network related tools will be of immense use in such an endeavor and issues relating to replication, checkpointing and rollback, network routing etc. will have to be looked into.

The work presented here has been more like laying down a foundation and it is expected that further research relating to more practical and higher level issues will be taken up in years to come.

REFERENCES

[Acce86] Accetta,M., Baron,R., Golub,D., Rashid,R., Tevanian,A., and Young,M., *MACH: A New Kernel Foundation for UNIX Development*, Technical Report, School of Computer Science, Carnegie Mellon University, August 1986.

[Ashi92] Ashish,S., *PAWAN : A MACH based UNIX System(I)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Bach86] Bach,M.J., *The Design of the UNIX Operating System*, Prentice-Hall Inc., Engelwood Cliffs, May 1986.

[Baro88] Baron,R.V., Black,D., Bolosky,W., Chew,J., Draves,R.P., Golub,D.B., Rashid,R.F., Tevanian,A.Jr., and Young,M.W., *MACH Kernel Interface Manual, Online MACH Documents (unpublished)*, School of Computer Science, Carnegie Mellon University, October 1988.

[Baru92] Baruah,M., *PAWAN : A MACH based UNIX System(III)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[Birr82] Birrell, A.D., Levin, R., Needham, R. M., and Schroeder M.D., *Grapevine: An Exercise in Distributed Computing*, Communications of the ACM April 1982, Vol 25, No 4.

[Birr84] Birrel,A.D., and Nelson,B.J., *Implementation of Remote Procedure Calls*, ACM Transactions on Computer Systems, Vol 1, Febraury 1984.

[Chas89] Chase,J.S., Amador,F.G., Lazowsk,E.D., Levy, H.M., and Littlefield,R.J., *The Amber System: Parallel Programming on a Network of Multiprocessors*, Communications of the ACM Mar 1989.

[Cher88] Cheriton,D.R., *The V Distributed System*, Communications of the ACM, 31(3), March 1988.

[Coop88] Cooper,E.C., and Draves,R.P., *C Threads*, Technical Report CMU- CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[Das89] Das,P.C., and Barua,G., *A Threads Facility for IITKIX*, CSI Journal of Computer Science and Informatics, Vol 19 (1989), No. 2.

[Drav89] Draves,R.P., Jones,M.B., and Thompson,M.R., *MIG --- The MACH Interface Generator*, Online MACH Documents (unpublished), School of Computer Science, Carnegie Mellon University, July 1989.

[Gopa92] Gopal,B., *PAWAN : A MACH based UNIX System(II)*, M.Tech Thesis, Indian Institute of Technology, Kanpur, April 1992.

[HCLHWM1] *VME ICP 16/8 Board Description*, Horizon II/III CE Reference Manual, Hindustan Computers Limited R&D Department, Document No. 301000150A.

[HCLHWM2] *SCSI Host Adapter Description and VME-QIC-02/R Hardware Reference*, Horizon II/III CE Reference Manual, Hindustan Computers Limited R&D Department, Document No. 301000150A.

[HCLHWM3] *VMEK20 CPU Board Description*, Horizon III CE Reference Manual, Hindustan Computers Limited R&D Department.

[INTEL] Intel 80386 Hardware Reference Manuals, Intel Corp., USA.

[Leff89] Leffler,S.J., McKusick,M.J., Karels,M.J., Quarterman,J.S., *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley Publishing Co., May 1989.

[LiHu89] Kai Li, and Hudak P., *Memory Coherence in Shared Virtual Memory Systems*, ACM Transactions on Computer Systems, Nov 1989, Vol 7, No 4.

[MCK20] Motorola MC68020 32-bit Microprocessor Hardware Reference Manuals, Motorola Inc., USA.

[Oppe83] Oppen,D.C., and Dalal,Y.K., *The Clearing House: A Decentralized Agent for Locating Named Objects in a Distributed Environment*, ACM Transactions on Information Systems, July 1983, Vol 1, No 3.

[Rash87] Rashid,R., Tevanian,A. Jr., Young,M., Golub,D., Baron,R., Black,D., Bolosky,W., and Chew,J., *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures*, Technical Report CMU-CS-87-140, School of Computer Science, Carnegie Mellon University, July 1987.

[Saty85] Satyanarayanan,M., Howard,J.H., Nichols,D.A., Sidebotham,R.A., Specter,A.Z., and West,M.J., *The ITC Distributed File System: Principles and Design*, Communications of the ACM, Dec 1985.

[Tane90] Tanenbaum et al., *Amoeba - A Distributed Operating System for the 1990's*, IEEE Computer, Vol 18. No.2, 1990.

[Teva87a] Tevanian,A. Jr., *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments*, PhD thesis, School of Computer Science, Carnegie Mellon University, December 1987.

[Teva87b] Tevanian,A. Jr., and Rashid, R., *MACH: A Basis for Future UNIX Development*, Technical Report CMU-CS-87-139, School of Computer Science, Carnegie Mellon University, Pittsburgh, June 1987.

[Teva87c] Tevanian,A. Jr., Rashid,R., Golub,D.B., Black,D.L., Cooper,E., and Young,M.W., *MACH Threads and the UNIX Kernel: The Battle for Control*, Technical Report CMU-CS-87-149, School of Computer Science, Carnegie Mellon University, August 1987.

[Teva87d] Tevanian,A. Jr., Rashid,R., Young,M.W., Golub,D.B., Thompson,M.R., Bolosky,W., and Sanzi,R., *A UNIX Interface for Shared Memory and Memory Mapped Files Under MACH*, Technical Report, School of Computer Science, Carnegie Mellon University, Pittsburgh, July 1987.

[Thom88] Thompson,M.R., *MACH Environment Manager, Online MACH Documents (unpublished)*, School of Computer Science, Carnegie Mellon University, July 1988.

[Walk83] Walker et al., *The LOCUS Distributed Operating System*, Proceedings of the Ninth Symposium on Operating Systems Principles, October 1983.

[Youn87] Young,M., Tevanian,A. Jr., Rashid,R., Golub,D., Eppinger,J., Chew,J., Bolosky,W., Black,D., and Baron,R., *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System*, Technical Report CMU-CS-87-155, School of Computer Science, Carnegie Mellon University, August 1987.